Système d'exploitation

II. Allocation de ressources

Kévin Perrot

Aix Marseille Université

2015

Ce cours est issu des supports de Jean-Luc Massat en L3 informatique à Luminy.

Table des matières

1 Allocations de ressources

1.1 Notion de ressources

Une ressource est un objet utilisable par un processus. Cette utilisation passe par le respect d'un mode d'emploi qui précise comment manipuler la ressource. Les ressources sont couramment *libres* ou *allouées*. Pour chaque ressource (ou famille de ressources) il

existe un *allocateur* qui a la charge de répondre aux requêtes de demande, de libération et éventuellement de réquisition.

Les ressources peuvent être *réquisitionnables* (CPU, mémoire) ou pas (unités de bande), *partageables* (mémoire, disques...) ou pas (imprimantes, unités de bande). Elles peuvent être *physiques* (celles que nous venons de citer, coupleurs...) ou *logicielles* (éditeur, canal). Dans ce dernier cas, c'est donc un programme qui est partagé entre plusieurs processus. La duplication de celui-ci en autant de copies qu'il y a de demandeurs est inconcevable du fait de la perte de taille mémoire que cela impliquerait. Les ressources logicielles dont le code ne change jamais (les données étant établies pour chaque demandeur) sont dites *réentrantes*. Les ressources peuvent également être *banalisées* si on dispose de plusieurs occurrences identiques d'une même ressource.

1.2 Objectifs et outils de l'allocation de ressources

Face à tous ces types de ressources, il est souhaitable de définir clairement les objectifs de l'allocation de ressources. Ces objectifs se retrouvent dans la plupart des allocateurs que nous avons ou que nous allons étudier.

L'OS doit être *équitable* dans l'allocation de ressources tout en respectant les *priorités*. En d'autres termes, pour un même niveau de priorité, les demandes doivent être traitées sans favoritisme excessif. La forme la plus simple de l'équité consiste à éviter la privation de ressource, c'est à dire l'attente infinie par un processus d'une ressource qu'il n'aura jamais. C'est notamment le cas si il y a interblocage entre plusieurs processus (nous verrons ce cas dans les sections suivantes). Finalement, l'OS doit également éviter la congestion c'est à dire la demande excessive de ressources. En d'autres termes, l'OS doit veiller à ne pas accepter les demandes quand le système est en surcharge.

Un modèle mathématique des files d'attente peut fournir aux « designers » de système des solutions efficaces au problème d'allocation de ressource. Les paramètres de ce modèle sont :

— la loi de distribution des instants d'arrivée,

— la loi de distribution des demandes de service,

— la politique de gestion de la file d'attente,

— l'absence ou la présence d'un mécanisme de réquisition.

2 Les interblocages

Le but principal du système dans un environnement multiprogrammé est le *partage des ressources* disponibles sur le site entre l'ensemble des processus. Or certaines de ces ressources étant non partageables, un processus possédant une telle ressource aura un contrôle exclusif sur celle-ci. Si l'on généralise cela à plusieurs processus et à plusieurs ressources on voit facilement apparaître les risques d'interblocage. Leur potentialité est liée aux conditions suivantes :

1. les ressources sont utilisées en exclusion mutuelle c'est à dire par un seul processus à la fois ;
2. chaque processus utilise simultanément plusieurs ressources qu'il acquiert au fur et à mesure de ses besoins sans nécessairement libérer celles qu'il possède déjà ;
3. les ressources ne peuvent être réquisitionnées ;

4. il existe un ensemble de processus $(p_0, p_1, \dots, p_n)$ tel que chaque p_i attend une ressource occupée par $p_i + 1$ et p_n attend une ressource occupée par p_0.

Après avoir présenté quelques exemples, nous étudierons dans les paragraphes qui suivent quelques méthodes employées pour prévenir, éviter, détecter et guérir les interblocages.

— La prévention est basée sur le principe de maintenir à chaque instant le système dans un état tel qu'aucun interblocage ne soit possible. Cette attitude est parfaitement efficace tant que l'on ne considère que l'aspect interblocage, mais en contre partie elle engendre un mauvais rendement d'utilisation des ressources. Néanmoins, ce genre de technique est très largement utilisé.

— Dans l'évitement de blocage, le but recherché est de rendre moins strictes les conditions imposées au système, comparativement à la prévention, afin de mieux utiliser les ressources. En fait, dans l'évitement, la possibilité de blocage existe à chaque instant, mais chaque fois que celui-ci s'approche, il est prudemment contourné.

— Les méthodes de détection se limitent à déterminer si un interblocage est apparu et si c'est le cas, quels sont les processus et les ressources qui sont impliqués. Ce travail étant fait, l'interblocage peut être traité et supprimé.

— Les méthodes de guérison sont utilisées pour « guérir » un interblocage en permettant à certains processus impliqués de terminer leur exécution afin de libérer les ressources qu'ils utilisent. En fait, ces techniques, la plupart du temps, consistent à supprimer un ou plusieurs des processus bloqués. Ceux-ci sont repris ensuite, généralement à partir du début, leur exécution précédente ayant été perdue.

2.1 Ressource unique

La plupart des risques d'interblocage dans un système sont dus aux ressources à accès unique. La figure 1 illustre ce type de configuration. Nous y voyons deux processus et deux ressources à accès unique. Une flèche allant d'une ressource à un processus indique que celui-ci détient celle-là ; une flèche allant d'un processus à une ressource signifie que celui-là est demandeur de celle-ci.

Nous avons donc dans le cas présent un interblocage puisque le processus A possède la ressource 1 et désire acquérir en plus la ressource 2 alors que celle-ci est détenue par le processus B qui réclame la ressource 1. Cette configuration bouclée est caractéristique des interblocages.

2.2 Interblocage dans un système de *spooling*

Rappelons que l'utilité d'un système de *spooling* est de ne plus assujettir l'exécution d'un programme à la lenteur de certains périphériques tels que l'imprimante. Une sortie sur un tel périphérique sera donc aiguillée vers un support d'accès beaucoup plus rapide (disque magnétique, par exemple) afin de libérer le programme. L'échange effectif avec le périphérique sera effectué ensuite, à partir du fichier *spool* constitué, via une unité d'échange.

Pour reprendre l'exemple de l'imprimante, on ne pourra tolérer qu'un programme qui tourne plusieurs heures au rythme de 100 lignes d'impression toutes les 10 minutes monopolise ce précieux périphérique durant tout ce temps. C'est la raison pour laquelle les fichiers *spool* ne seront imprimés qu'après achèvement des programmes correspondants.

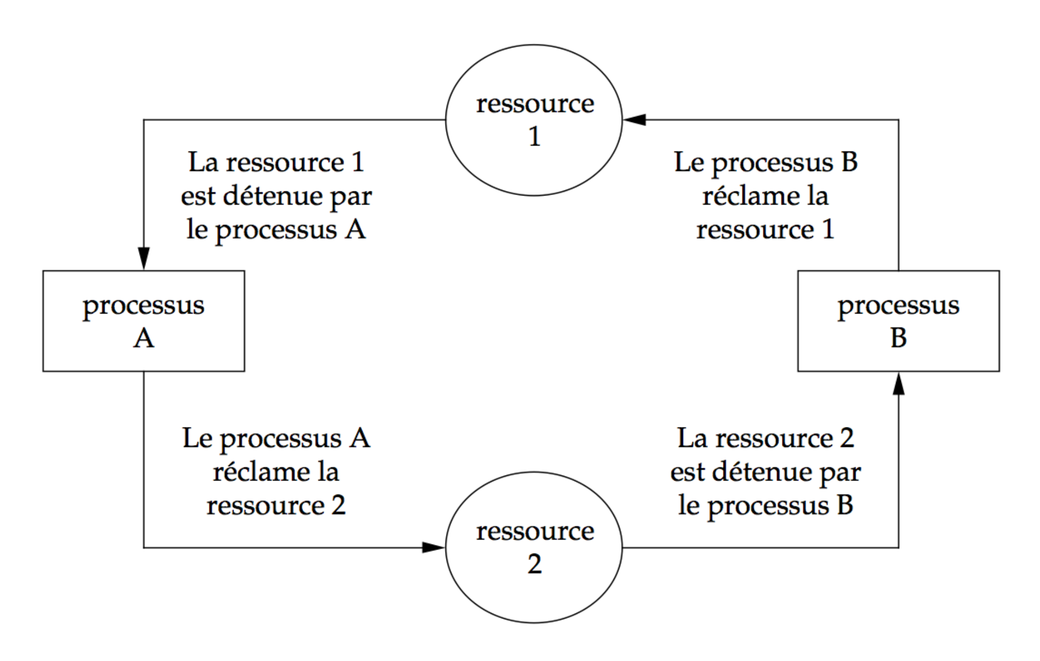


Figure 1 – Interblocage simple.

Dans ces conditions, le problème qui se pose est celui de la place prévue pour l'ensemble des fichiers *spool*. Le spectre de l'interblocage se dessine peu à peu. Il peut en effet arriver un moment où la zone de *spool* étant saturée, plus aucun processus ne puisse opérer des sorties, mais aucun n'étant achevé, la zone *spool* ne peut se vider !

Que faire à ce moment là ?

— Supprimer un processus (et perdre toute l'exécution) pour récupérer la place qu'il occupait ? Qui nous assure que l'on ne sera pas obligé ensuite d'en supprimer un deuxième puis un troisième ? …

— Commencer à imprimer les sorties d'un des processus ? Lequel choisir ? Monopolisera-t-il l'imprimante longtemps ? Le principe lui-même est-il acceptable ?

— Aurait-on pu prévenir cette situation en interdisant tout accès à un nouveau fichier *spool* (c'est à dire en fait tout nouveau travail) dès que l'occupation avait atteint un certain taux ?

2.3 Autre forme de blocage : la famine

Dans tout système où des processus sont en attente pendant que des ressources sont allouées et en particulier le processeur, il est possible que l'activation d'un processus soit indéfiniment retardée alors que les autres sont servis. Cette situation de *famine* est aussi préjudiciable qu'un blocage.

Lorsqu'une ressource est allouée sur la base de priorités, il se peut qu'un processus reste en attente pendant qu'une suite ininterrompue de processus de priorité plus élevée ont la préférence. Or le propre d'un système est d'être à la fois équitable et efficace envers les processus en attente. On verra à plus loin comment on peut accroître la priorité au fur et à mesure que le temps d'attente augmente. Ce type de technique présenté pour la ressource processeur peut être appliqué pour n'importe quelle ressource.

3 La prévention de blocages

C'est assurément la technique la plus utilisée par les *designers* de système. Nous allons voir ici quelques unes des méthodes proposées en considérant leurs effets à la fois sur les utilisateurs et sur le système en particulier du point de vue des performances.

Havender proposa de *mettre en défaut les conditions nécessaires d'interblocage* en imposant des contraintes aux processus :

— Tout processus doit *annoncer* les ressources qui vont lui être nécessaires et ne *démarrer* que lorsque toutes sont disponibles.

— Si un processus à besoin d'une ressource supplémentaire, il doit libérer celles en sa possession et faire une nouvelle demande incluant la nouvelle.

— Les ressources sont classées par type dans un ordre linéaire auquel devra se soumettre tout processus pour ses demandes d'allocation.

— Il est à noter que chacune des stratégies proposées ci-dessus a pour but de mettre en défaut une des conditions nécessaires d'interblocage sauf la première. En effet, nous voulons nous réserver le droit de disposer de ressources dédiées.

3.1 échec à la condition d'attente

La première stratégie d'Havender impose que toutes les ressources nécessaires à un processus soient libres avant qu'il puisse commencer ; ce sera donc du « tout ou rien ». En aucune façon, les ressources utiles ne pourront être réservées jusqu'à ce que toutes étant libres, l'exécution puisse commencer : elles devront êtres toutes libres simultanément ! La deuxième condition nécessaire est ainsi trivialement mise en défaut... mais à quel prix ! Quel gaspillage de ressources ! Supposons qu'un programme dont l'exécution dure plusieurs heures ait besoin la plupart du temps de un ou deux dérouleurs de bandes sauf pendant un court instant, en fin d'exécution, où dix unités lui sont nécessaires. En appliquant cette stratégie, le processus devra monopoliser les dix dérouleurs pendant toute son exécution. De plus, il devra attendre qu'ils soient tous libres avant de pouvoir être initialisé, ce qui risque d'être long ! Nous nous trouvons devant un risque flagrant de famine.

Une solution utilisée pour remédier à ce gros défaut consiste à opérer par étapes lorsque, comme dans l'exemple que nous venons de voir, le programme s'y prête. Dans ces conditions, l'allocation des ressources se fera elle aussi par étape, ce qui réduit considérablement la sous-utilisation des ressources mais engendre un coût d'exploitation plus élevé.

3.2 échec à la condition de non-réquisition

Supposons que le système autorise un processus à conserver les ressources qu'il détient alors qu'il opère une nouvelle demande. Tant que les ressources supplémentaires demandées sont libres, le blocage n'apparaît pas. Mais si nous arrivons dans un schéma montré dans la figure 1, nous sommes en situation d'interblocage.

Havender préconise en pareil cas d'imposer à un processus demandeur de libérer les ressources qu'il détient pour ensuite les redemander en y ajoutant la nouvelle. Cette stratégie met en échec la troisième condition nécessaire. Mais là encore, à quel prix ? Lors de la libération obligatoire des ressources détenues, tout un travail peut être perdu

(bonjour les performances système) ! Si cela se produit peu souvent, c'est tolérable, mais si c'est fréquent ce peut être catastrophique : en particulier des travaux prioritaires et/ou à échéance risquent de voir leur statut sérieusement remis en cause, sans parler des risques évidents de famine.

3.3 échec à la condition d'attente circulaire

C'est le but recherché par la troisième stratégie proposée par Havender. Chaque type de ressource ayant un numéro, tout processus ne pourra effectuer ses requêtes que par ordre croissant dans ces types. Cette stratégie a été implantée dans de nombreux systèmes, mais non sans difficulté.

— Les diverses ressources étant requises au moyen de leur numéro d'ordre, l'ajout d'une nouvelle ressource sur un site nécessite la modification de tous les programmes. La portabilité est nulle.

— Le numéro d'ordre alloué aux diverses ressources doit refléter l'ordre d'utilisation de la plupart des programmes susceptibles d'être exécutés sur le site. Si d'aventure un programme ne respecte pas cet ordre « canonique », les ressources doivent être acquises éventuellement longtemps avant leur utilisation effective. D'où un gaspillage.

— Les systèmes tendent de plus en plus aujourd'hui à respecter la contrainte de *convivialité*. Le moins que l'on puisse dire est que cette stratégie ne répond pas à cette attente.

4 Évitement d'interblocage (l'algorithme des banquiers)

Dans un système où les risques d'interblocage existent, il est toujours possible de l'éviter en prenant les précautions nécessaires à chaque allocation de ressource. La technique la plus connue est sans doute l'algorithme des banquiers de Dijkstra, ainsi nommée à cause de la grande prudence de ceux-ci en matière de prêts [1] : « On ne se lâche pas des pieds sans se tenir des mains ! ».

Partons du principe que l'OS connaît parfaitement l'état de l'allocation de ressources aux processus. Plus précisément, les données suivantes sont considérées comme disponibles :

— $dispo[i]$ nombre de ressources R_i disponibles sur le système,

— $max[i, j]$ nombre maximum de ressources R_i utilisables par le processus P_j,

— $alloc[i, j]$ nombre de ressources R_i couramment allouées au processus P_j (par définition nous avons donc $alloc[i, j] \leq max[i, j]$).

Un processus P_j peut s'exécuter si et seulement si, pour toute ressource R_i, nous avons $max[i, j] - alloc[i, j] \leq dispo[i]$.

Un ordre d'exécution $P_1, P_2, \ldots, P_n$ est dit *sain* si et seulement si les processus $P_1, P_2, \ldots, P_n$ peuvent s'exécuter jusqu'à leur terme les *uns après les autres* dans cet *ordre*. Bien entendu l'exécution d'un processus P_k implique la *libération* par P_k des ressources qu'il a utilisées. Un système est dit *sain* si il existe un ordre d'exécution sain des processus. Si un système est sain, alors il ne peut pas y avoir d'interblocages. Par contre, si le système n'est pas sain, un interblocage peut apparaître mais ce n'est pas une obligation.

1. Disons que c'était peut-être vrai à l'époque, de nos jours on aurait certainement choisi un autre nom. . .

En résumé, l'apparition d'un état non sain n'implique pas pour autant qu'il y aura inévitablement un interblocage. La seule chose que cela implique est qu'une séquence défavorable d'événements peut conduire à un interblocage.

En se basant sur cette propriété, l'OS face à une demande d'allocation de la ressource R_i au processus P_j, applique l'algorithme suivant :

1. les annonces sont elles respectées (c-à-d $alloc[i, j] < max[i, j]$) ?
2. si j'alloue R_i à P_j l'état obtenu est-il sain ?
3. dans l'affirmative je réalise l'allocation ; sinon je suspends le processus P_j qui ne peut donc pas continuer son exécution.

Le principe de l'algorithme des banquiers est de refuser toute requête ayant pour effet de mettre le système dans un état non sain. En résumé :

— les conditions d'exclusion mutuelle, d'attente et de non réquisition sont autorisées ;

— les processus doivent annoncer leurs besoins en ressources ;

— ils peuvent conserver les ressources en leur possession tout en réclamant des ressources supplémentaires ;

— il n'y aura pas de réquisition ;

— afin d'aider le système, les ressources seront demandées une à une ;

— si une requête n'est pas honorée, le processus demandeur conserve néanmoins les ressources en sa possession et attend un temps fini jusqu'à ce qu'il obtienne satisfaction ;

— seules les requêtes laissant le système dans un état sain sont honorées. Dans l'éventualité contraire, le processus demandeur devra attendre (le système étant toujours dans un état sain, toutes les requêtes pourront être satisfaites tôt ou tard).

Cet algorithme semble donc intéressant et à tout le moins plus convivial et d'un meilleur rendement que les stratégies de prévention proposées par Havender. Toutefois, nous allons voir qu'il contient de nombreuses faiblesses qui font que les concepteurs de systèmes lui préfèrent d'autres approches.

— L'algorithme présuppose et s'appuie totalement sur le fait que le nombre de ressources est invariant. Or il est bien évident que des problèmes de maintenance du matériel ou tout simplement des pannes peuvent mettre en défaut ce postulat.

— Il présuppose aussi que chaque utilisateur annonce le maximum de ressources utilisées. Or à l'heure actuelle, la convivialité grandissante des systèmes fait que rares sont les utilisateurs connaissant précisément les ressources dont ils ont besoin.

— L'algorithme garantit que les requêtes pourront être satisfaites... dans un temps fini (!) Voilà qui est rassurant mais guère suffisant !

— Réciproquement, il impose aux processus de restituer les ressources... au bout d'un temps fini. Là encore, on pourrait s'attendre à un peu plus d'exigence !

5 Détection des interblocages

Les algorithmes de détection sont utilisés dans des systèmes où les trois premières conditions nécessaires sont autorisées et ont pour but de déterminer s'il y a attente circulaire. L'utilisation de ces algorithmes entraîne un coût d'exploitation non négligeable.

5.1 Graphes d'allocation de ressource

L'utilisation de graphes orientés, représentant les allocations et requêtes de ressources, facilite la détection des blocages. Dans les schémas qui vont suivre, les carrés représentent des processus et les cercles des classes de ressources identiques. Les petits cercles contenus dans ces derniers représentent le nombre de ressources de chaque classe. La figure 2 montre les relations pouvant être représentées dans un graphe d'allocation et de requête des ressources. Ces graphes sont modifiés au cours du temps à chaque nouvelle allocation ou libération de ressource sur le site. S'il advient qu'une ressource d'un type donné soit hors service (pour une cause quelconque), cela se traduira dans le graphe par la suppression d'un petit cercle dans le grand cercle correspondant au type en question.

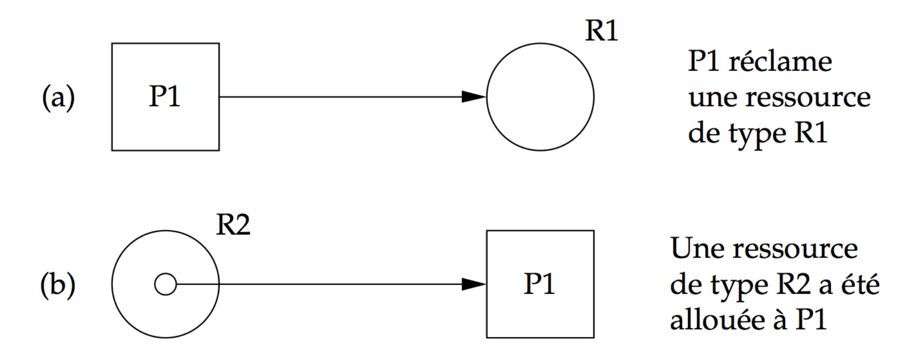


FIGURE 2 – Graphe de requête et d'allocation de ressource

Afin de déterminer s'il est en situation de blocage, le système devra procéder à une réduction du graphe.

5.2 Réduction d'un graphe d'allocation de ressource

Si les requêtes d'un processus peuvent être satisfaites, on dit que le graphe est réduit par ce processus (cela signifie que l'on considère le graphe comme si le processus s'était achevé, libérant ainsi les ressources qu'il détenait). Cela se traduit par la suppression des fléches provenant de ressources et aboutissant à ce processus et de celles partant de ce processus vers d'autres ressources.

Si un graphe peut être réduit par tous ses processus, il n y a pas de blocage. Dans le cas contraire, les processus irréductibles constituent l'ensemble des processus en interblocage.

La figure 3 montre les diverses étapes de réduction d'un graphe permettant d'aboutir à la conclusion qu'il n'y a pas d'interblocage.

L'ordre dans lequel les réductions se font est sans importance : le résultat sera toujours le même.

6 Guérison d'interblocage

Une fois que le système a déterminé qu'il y avait interblocage, il doit le guérir en supprimant une ou plusieurs des conditions nécessaires. Habituellement, un ou même plusieurs processus perdront pour ce faire tout ou partie du travail déjà accompli. Mais mieux vaut cela que le maintien de l'interblocage. La guérison est rendue difficile pour plusieurs raisons :

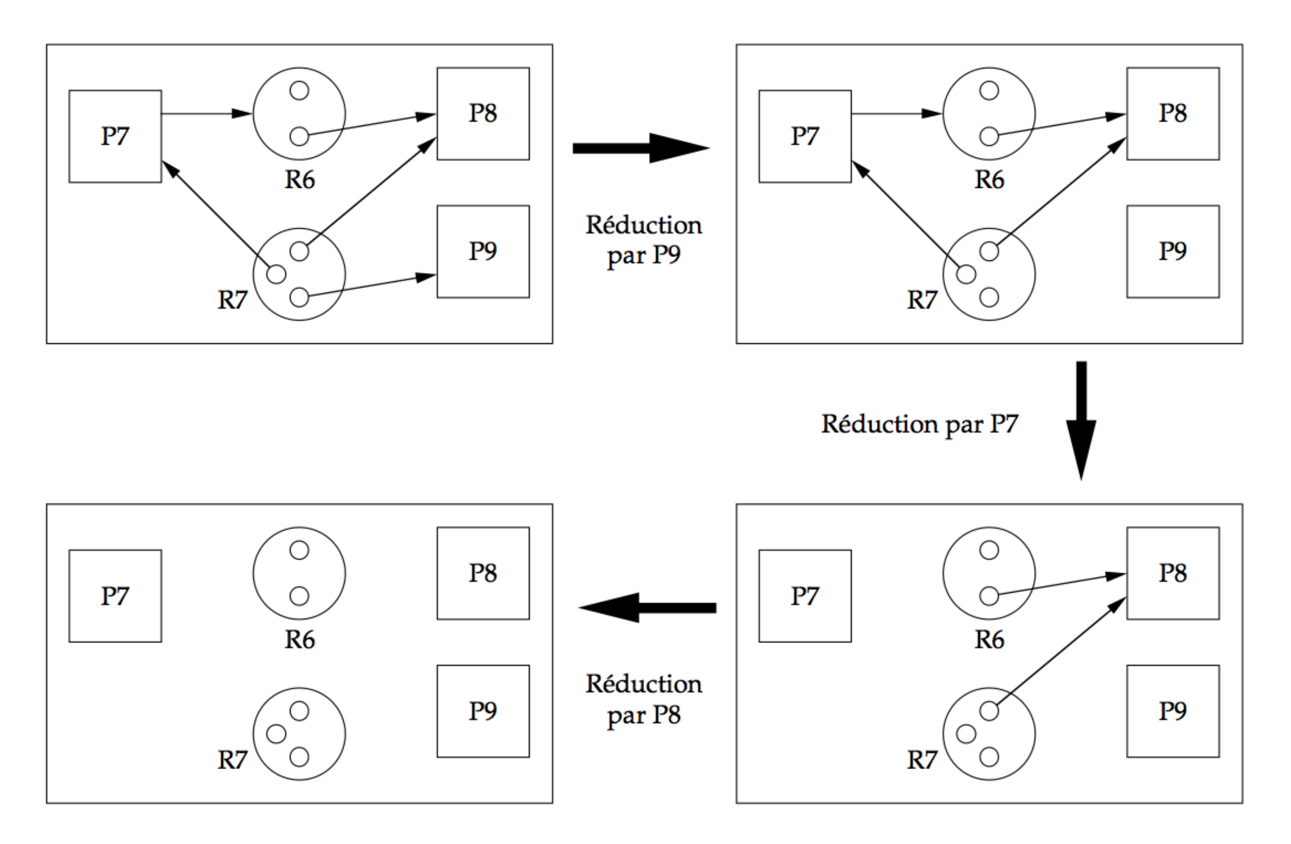


FIGURE 3 – Réduction d'un graphe d'allocation.

— Tout d'abord, nous venons de le voir, la détection de l'interblocage n'est pas chose aisée, et le système peut ne pas s'en apercevoir tout de suite.

— La plupart des systèmes n'ont guère de facilité pour suspendre indéfiniment un processus, l'enlever du système et le reprendre plus tard. En particulier, cela est hors de question pour les processus temps réels.

— Même si ces possibilités existaient, elles entraîneraient un coût d'exploitation prohibitif et nécessiteraient les compétences d'un opérateur attentif, ce qui n'est pas toujours possible !...

— La guérison d'un interblocage de proportions modestes peut être opérée avec un coût raisonnable ; mais si l'on est en présence d'un interblocage de grande envergure (faisant intervenir plusieurs dizaines ou même centaines de processus), la quantité de travail sera énorme.

La guérison passe quasi incontournablement par la destruction d'un processus afin de récupérer les ressources qu'il possédait pour permettre aux autres processus de s'achever. Quelquefois la destruction de plusieurs processus s'impose pour récupérer un nombre suffisant de ressources. Aussi le terme de guérison semble ici un peu exagéré, mais s'adapte parfaitement à la conception occidentale de la médecine qui s'attache surtout à la symptomatique des pathologies. (On peut faire la comparaison avec une amputation d'un membre atteint d'artérite : le patient s'estime-t-il guéri ?)

L'ordre dans lequel les processus vont être supprimés est très important. Va-t-on chercher à minimiser leur nombre ? Va-t-on considérer la quantité de travail déjà accomplie afin de réduire la perte de rendement ? Va-t-on considérer les priorités des processus ? Va-t-on choisir les processus victimes parmi ceux pour lesquels le retrait des ressources n est pas fatal à 1 exécution (afin de seulement les suspendre, le temps de « guérir », pour

ensuite les reprendre en l'état, sans perte de travail) ?

Et le temps d'exploitation de tout cela ? Et si cela arrive sur un site sur lequel se déroule une application temps réel très délicate (surveillance de raffinerie, de centrale atomique. . .). Voilà autant de questions qui donnent à réfléchir sur la conception des systèmes de demain et qui, loin d'être résolues de façon satisfaisante, occasionnent quelques « nuits blanches » aux concepteurs d'aujourd'hui.

7 Et demain ?

Nous venons d'avoir un aperçu de ce qui se faisait (ou pourrait se faire) aujourd'hui. En fait, ce sont les méthodes de prévention carrées, brutales, mais efficaces et sans risque qui sont le plus souvent employées, le blocage étant encore tout à fait occasionnel.

Dans les systèmes futurs, les interblocages devront être traités de façon systématique et efficace pour plusieurs raisons :

- Les systèmes s'orienteront de plus en plus vers des opérations parallèles asynchrones, abandonnant les schémas séquentiels. Les bancs de processeurs seront monnaie courante, autorisant un parallélisme énorme.
- L'allocation des ressources sera dynamique. La convivialité grandissante des systèmes fera que l'on utilisera ce que l'on voudra quand on le voudra.
- De plus en plus, les données seront assimilées à des ressources. En conséquence, la quantité des ressources que devra gérer un système atteindra une taille gigantesque.

On peut donc imaginer que ce problème, tout en devenant de plus en plus important et de plus en plus difficile à traiter, trouvera néanmoins dans les technologies et les structures futures des solutions efficaces.